PARALLEL IMPLEMENTATIONS OF PROBABILISTIC LATENT SEMANTIC ANALYSIS ON GRAPHIC PROCESSING UNITS

BY

HANG CHEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

ChengXiang Zhai

ABSTRACT

Probabilistic Latent Semantic Analysis (PLSA) has been successfully applied to many text mining tasks such as retrieval, clustering, summarization, etc. PLSA involves iterative computation for a large number of parameters and may take hours or even days to process a large dataset, thus speeding up PLSA is highly motivated in the domain of text mining. Recently, the general purpose graphic processing units (GPGPU) have become a powerful parallel computing platform, not only because of GPU's multi-core structure and high memory bandwidth, but also because of the recent efforts devoted into building a programming framework to enable developers to easily manipulate GPU's computing power. In this paper, we introduced two methods to parallelize and speed up PLSA via GPGPU. Related issues are addressed including workload balance, block-thread layout, memory and data access optimization, etc. The GPU in use is NVidia GTX480 (costs $450 in market). Experimental results show that our methods can process 300,000 documents in 12 seconds which is a 33x speedup compared with traditional PLSA implementation running on 3.0GHz Intel Xeon CPU. The significant speedup can bring researchers in the text mining domain brand new experience.

To my parents and my girlfriend Qing, for their constant love and support.

ACKNOWLEDGMENTS

Time flies. The day I came to the school is still clear. During the two and a half years I have learned a lot. Thank our school, our C.S. Department for its decent and inspiring research atmosphere, from which I realized the significance of research work and through what way it should be done. Thank all teachers who have taught me, every one of them expanded my knowledge and horizon as well as affected me by their charming characteristics. Thank my supervisor Cheng especially, for his caring attitudes and profound stewardship. Also thank my peers who are all so outstanding that I always need to keep pushing myself to catch up and this to great extent released my potential. GPU resources used in paper are from the AC-cluster at NCSA. After all, thank my parents and my girlfriend Qing for their love and support.

TABLE OF CONTENTS

INTRODUCTION

Probabilistic Latent Semantic Analysis (PLSA) is a popular topic modeling algorithm for its effectiveness and ease-of-use, and it has been successfully applied to many text mining tasks such as retrieval, clustering, summarization, etc. Unfortunately, PLSA requires considerable amount of computation and may take hours or even days to process a large dataset [1]. As a result, speeding up PLSA can be bliss for researchers in the domain of text mining.

To speed up computations in a general and scalable manner, people mainly seek for parallel infrastructure, because the frequency of computing units becomes bounded and a single core can no longer be faster. Under this trend, Graphic Processing Units (GPU) recently attracts huge attention for its hundreds-of-core structure and high memory bandwith, as it was designed to handle high-granularity graphics-related applications like rendering and simulation where many independent workloads are simultaneously dispatched to processor elements. Also with the development of its programmability, GPU gradually becomes a general platform for parallel computing, which is known as general purpose GPU (GPGPU) [2]. In this thesis, we explore a novel way of speeding up PLSA by exploiting GPGPU.

We proposed two methods which emphasize different speedup factors. The Matrix-based method better exploits the GPU's parallism and high memory bandwidth while the Doc-distribution method better accommodates the nature of PLSA and is more memory efficient. Experimental results show that our methods takes only 12 seconds to process 300,000 documents which is 33x speedup comparing to a powerful CPU. This is also a much more significant speedup than the state-of-the-art speedup method which is CPU-based under similar cost [3].

Rest of the thesis is organized as follows: Chapter 2 introduces some related work; Chapter 3 goes through the algorithm of PLSA; Chapter 4 introduces necessary background about GPU and NVdia's CUDA programming

framework; Chapter 5 describes our methods of parallelizing and speeding up PLSA on CUDA in detail; Chapter 6 shows the experimental results with discussions; Chapter 7 gives some future directions and concludes the thesis.

RELATED WORK

In the domain of machine learning and text mining, parallelization and speedup of various kinds of computing algorithms have attracted substantial attention from researchers [3] [4] [5]. The emerging GPGPU technology makes this topic even more popular recently [6] [7] [8] [9] [10]. For example, GPU-based K-means clustering parallelization has been thoroughly studied and various methods were proposed in which up to 40x speedup can be achieved [6] [7] [8]. Besides, Yu et. al. applied GPU in Gibbs sampling for motif finding and achieved 10x speedup [9]. Yan et. al. proposed a parallel inference method for Latent Dirichlet Allocation (LDA) on GPU and achieved 20x speedup [10]. From an engineering point of view, each algorithm's GPU implementation requires a unique design regarding to multiple issues including workload distribution, block-thread layout, memory optimization, etc, and the design should match the GPU hardware and driver functionality in use. As a result, previous GPU methods of other algorithms cannot be directly applied to PLSA speedup, which is the goal of our study.

There have been some work on speeding up PLSA. Hong et. al. proposed a CPU-based parallelization algorithm for PLSA and made 6x speedup on 8-core CPU machines [3]. None of the existing methods has ever exploited GPU, which has hundreds of cores running at over 1GHz (rather than a top-end 8-core CPU machine) and high memory bandwidth. We believe that proper utilization of its characterics can well serve our speedup purpose, which is proved by experiments. In summary, to the best of the author's knowledge, no one has used GPU to implement PLSA or speeded up PLSA to a level as we do.

OVERVIEW OF PLSA

Probabilistic topic models are based on the fundamental idea that documents are mixture of topics, where a topic is represented by a multinomial distribution of words, i.e., a unigram language model. Probabilistic latent semantic analysis (PLSA) was introduced by Hofmann [11]. A document d is regarded as a sample of the following mixture model [1].

$$P(w|d) = \Sigma_{j=1}^{K} \phi_w^{(j)} \theta_j^{(d)} \tag{1}$$

where $\theta_j^{(d)} = P(z = j|d)$ is the multinomial distribution over topics for document d, $\phi_w^{(j)} = P(w|z = j)$ is the multinomial distribution over terms for topic j, K is the number of topics [1]. The word-topic distribution ϕ and topic-document distribution θ can be estimated using the Expectation-Maximization (EM) algorithm by maximizing the (log) likelihood that the collection D is generated by this model [1]:

$$logP(D|\phi, \theta) = \Sigma_{d \in D} \Sigma_{w \in W} \{c(w, d) log \Sigma_{j=1}^{K} \ \phi_w^{(j)} \theta_j^{(d)}\} \tag{2}$$

where c(w, d) is the number of times word w occurs in document d. In E-step, hidden variable z is estimated based on the model parameters at the previous iteration:

$$P(z_{d,w} = j) = \frac{\phi_w^{(j)} \theta_j^{(d)}}{\Sigma_{j'=1}^{K} \ \phi_w^{(j')} \theta_{j'}^{(d)}} \tag{3}$$

Then in M-step, update the parameters given $P(z_{d,w} = j)$.

$$\theta_j^{(d)} = \frac{\Sigma_{w \in V} \ c(w, d) P(z_{d,w} = j)}{\Sigma_{j'} \ \Sigma_{w \in V} \ c(w, d) P(z_{d,w} = j')} \tag{4}$$

$$\phi_w^{(j)} = \frac{\Sigma_{d \in D} \ c(w, d) P(z_{d,w} = j)}{\Sigma_{w' \in V} \ \Sigma_{d \in C} \ c(w', d) P(z_{d,w'} = j)} \tag{5}$$

We can see that in E-step, values of $P(z_{d,w} = j)$ are not mutually affected thus can be computed in a parallel manner. Similarly, in M-step, given $P(z_{d,w} = j)$, all $\theta_j^{(d)}$ and $\phi_w^{(j)}$ can be computed at the same time. Also in PLSA, because the number of documents and terms are usually very large, the hundreds-of-cores structure of GPU can be well exploited thus it would likely outperform CPU-based parallelization which involves at most tens of cores (under similar cost). Moreover, by writing the above equations into matrix operations and let GPU directly handle matrices, memory schema can be further optimized and GPU's super-fast in-block memory can be utilized. Detail of the methods will be presented in Chapter 5.

OVERVIEW OF GPU AND CUDA

GPU can be viewed as a set of multiprocessors executing concurrent threads in parallel. Under CUDA, which is NVidia's parallel computing GPU architecture, threads are grouped into thread blocks and execute the same set of instructions in parallel. Within one block threads can be synchronized at any execution point, certain execution order of threads within a block is not guaranteed. Blocks are further grouped into grids, communication or synchronization between blocks cannot be achieved. One or more blocks are directly mapped to a multiprocessor but order of running is not defined. Threads and blocks are organized in up to three and two dimensional manners respectively. Each thread and block is assigned an ID depending on its position within the chip grid which can be accessed at run time. Each thread on the GPU executes the same procedure known as kernel [2].

Each GPU chip has a global memory of few gigabytes, and each block has its internally shared memory. Shared memory access inside block is extremely fast comparing to global memory access. A good design of implementing an algorithm on GPU involves deriving balanced workload distribution over a reasonable block-thread layout together with optimization of usage of shared memory, for the purpose that both multi-core parallelism and wide memory bandwidth can be at best exploited [2].

The CUDA SDK gives developer easy to use tools and full integration with any C++ compilers. Code for GPU is written in a subset of C with some extensions and can coexist with CPU (host) code in the same source file. The host code is responsible for setting up the layout of blocks and threads as well as uploading data to GPU. Kernel execution is performed asynchronously, while primitives to synchronize between CPU and GPU code are available [2].

With the fast development of GPU's hardware and software computability, old issues including "atomic adding" and "double number" are gradually solved and optimized, memory size got larger and is now sufficient for most

datasets under our methods. It is time for us text mining researchers to feel its power.

GPU-BASED PARALLEL METHODS FOR PLSA

In this chapter, we proposed two methods to parallelize and speed up the PLSA algorithm. With the equations from Chapter 3 in mind, the problem of PLSA is defined as follows.

Problem definition: Given DN documents $\{d_1, d_2, ..., d_{DN}\}$; TN terms $\{t_1, t_2, ..., t_{TN}\}$; K topics $\{z_1, z_2, ..., z_K\}$. Two sets of parameters need to be estimated as described: $\theta_j^{(d)} = P(z = j|d)$ and $\phi_w^{(j)} = P(w|z = j)$. The number of cores on the chip is PN. Two methods are proposed and implemented in this thesis.

1 Doc-distribution Method

As discussed in Chapter 3, we would naturally want to distribute the computations of all $P(z_{d,w} = j)$ onto different processors. In the first method we proposed, each processor handles a set of documents. And for each document, the assigned processor goes through all the terms in it, generate $P(z_{d,w})$ (as Eq.3) and then make certain increment to the numerator of Eq.4 and Eq.5, and finally $\phi_w^{(j)}$ and $\theta_j^{(d)}$ are normalized.

In particular, we assigned 256 threads for each block, i.e., one block handles 256 documents at a time. This number is suggested by some CUDA best practices and is further tuned and proved in our experiments. One or more blocks are mapped to a hardware multiprocessor while time sharing governs the execution order.

Computation complexity: while the traditional method takes $O(DN \times tn_{avg} \times K)$ where tn_{avg} is the average number of terms a document has, the time complexity of the Doc-distribution method is $O(\frac{DN}{PN} \times tn_{avg} \times K)$. However, the ideal 'linear' is hard to achieve due to several reasons.

The major issue of this method turned out to be that since $\phi_w^{(j)}$ and $\theta_j^{(d)}$

are both global variables in the incrementing phase and are updated by all threads, the update operations have to be "atomic add" of floating number which is costly (actually this functionality is not supported in CUDA, for those who are curious, we achieved this by utilizing $atomicCAS()$ function which does an atomic "check-and-swap" operation. The code is listed in Appendix). Another issue is that different documents may have very different numbers of terms, thus the workload cannot be evenly distributed inside each block. However, because the number of documents is much greater than the number of blocks and there are hundreds of multiprocessors to accommodate, this is not a serious issue under CUDA's dynamic scheduling schema. We will demonstrate this by presenting the experimental results in Chapter 6.

2 Matrix-based Method

To cope with the issue of simultaneous access of global memory and workload distribution in the last method, we proposed another method. The idea is to granulize the parameters so that global memory conflicts are avoided and workload can be perfectly balanced.

The natural way to achieve that is to calculate $P(z_{d,w} = j)$ first, put this 3D matrix in memory and then update $\phi_w^{(j)}$ and $\theta_j^{(d)}$. However, P being a 3D matrix requires huge memory, which can easily exceed the device memory capacity of current high-end GPUs under large datasets, so saving P is not practical.

One way to work around is to write PLSA into matrix operations such that the 3D parameter P is hidden throughout the process. The Equations 3, 4, 5 are re-written into matrix operations as below [12]:

$$\Theta = \Theta. \times (\Phi' \times (C./(\Phi \times \Theta))) \tag{6}$$

$$\Phi = \Phi. \times ((C./(\Phi \times \Theta)) \times \Theta') \tag{7}$$

where Θ is the matrix form of θ in Eq.4, i.e., $\Theta_{j,d}$ equals $\theta_j^{(d)}$; and Φ is the matrix form of ϕ in Eq.5, i.e., $\Phi_{w,j}$ equals $\phi_w^{(j)}$. C is the doc-term count matrix which has TN rows and DN columns. '$\times$' is the matrix-multiplication operator. '$.\times$' is dot-multiplication operator, i.e., to multiply the two elements at the same location in two matrices and put the product in that

location in the resulting matrix, and two matrices have to be of the same size. './' is dot-division operator works the same way as '.×' except it is doing 'divide' instead of 'multiply'. The normalization procedure of Θ and Φ need to be added after updating each of them. Eq. 6 and 7 together with the normalization procedure works the way same as Eq. 3,4,5.

In this method, all the computations are fully parallelizable (matrix multiplication, dot division and multiplication) [2] except for the normalization procedures of Θ and Φ (which is block-level parallelizable). Another advantage of this method is that for matrix multiplication operations which take up the part of computation, GPU's fast in-block memory can be utilized to further improve the performance (The trick is to break up the execution of the kernel into phases so that the data accesses in each phase are focused on one tile of the matrix [13]).

For all the matrix operations, the thread number of each block is set to be 16×16, as related best practice suggests. Computation is divided into several stages in terms of the resulting matrix size so that the corresponding block-thread layout can best exploit parallism. For example, the computation of $\Theta = \Theta. \times (\Phi' \times (C./(\Phi \times \Theta)))$ takes three stages:

- In the first stage, $C./(\Phi \times \Theta)$ is computed. The computation process contains one "matrix-multiplication" ($TmpM1 = \Phi \times \Theta$) and "one matrix-dot-division" ($TmpM2 = C./TmpM1$). Temporary matrices like 'TmpM1' in the equations are introduced only for illustration purpose. Because the size of the resulting matrix is $TN \times DN$, $\frac{TN}{16} \times \frac{DN}{16}$ blocks are assigned and each block has 256 threads as described.

- In the second stage, Φ is transposed into Φ'. Because the size of Φ is $TN \times K$, $\frac{TN}{16} \times \frac{K}{16}$ blocks are assigned.

- The third stage does the rest which includes one "matrix-multiplication" ($TmpM3 = \Phi' \times TmpM2$) and one "matrix-dot-multiplication" ($\Theta = \Theta \times TmpM3$). Because size of Θ is $K \times DN$, $\frac{K}{16} \times \frac{DN}{16}$ blocks are assigned.

- Note that the size of all related matrices are much higher than PN even for just median-size datasets, thus all processors will have job running throughout the algorithm and parallism is fully exploited.

Computation complexity: The computation complexity of the Matrix-based method is $O(\frac{DN \times TN}{PN} \times K)$.

The major issue of the Matrix-based method is that bag-of-words representation of documents usually makes the matrix extremely sparse. However, this method cannot take advantage of such sparsity, i.e., TN is much larger than tn_{avg}, which saturates the parallism to certain extent. One optimization for this issue can be using singular value decomposition (SVD) to densify and reduce the dimension of the matrix. The second issue is that although the giant matrix P is hidden, matrix C ($DN \times TN$ in size) still needs to be stored, which takes large memory and reduces the practicality. This issue can be solved by utilizing GPU's data streaming mechanism [15] so that the GPU memory is backed up by large CPU memory, and memory transfer operations are hidden behind the computation time so that the speedup effect is not degraded. We will explore these techniques in our future work.

EXPERIMENTS AND ANALYSIS

We set up series of experiments to compare the efficiency of running PLSA with our methods on GPU-supported machines and the baseline setting: running the most dominant PLSA implementation on CPU-based machines. This baseline is also adopted by other speedup methods.

1 Machines

The GPU we used is GeForce GTX480, which has 480 on-chip multiprocessors, each of which running at 1.4GHz. It has 1.5GB on-board device memory. The CPU in comparison is Intel Xeon CPU running at 3.0GHz with 4G memory.

2 Datasets

The datasets for evaluation are from UCI Machine learning Repository which are widely adopted in the research domain. The two chosen text datasets are representative in size: one median-size dataset and one large-size dataset [14]. Some statistics about the datasets is shown in Table 1.

Dataset	KOS	NYT
Number of documents	3430	300,000
Number of terms	6906	102,660
Number of doc-term pairs	353,160	69,679,427

Table 1: datasets used in the experiments

3 Speedup

The CPU implementation follows the Lemur's PLSA implementation in which redundant computation is reduced to its best. The validity of GPU's running results is proved by pre-setting the same values for parameter variables on both GPU and CPU, and confirming that generated results are exactly equal. We set K to be 50 which is a commonly used value. All parameters are double precision numbers to ensure accuracy. For the median-size dataset (KOS), the baseline setting (CPU) runs 1.56s for one iteration on average, while the Doc-distribution method (GPU) takes 0.140s and the Matrix-based method (GPU) takes 0.098s. For the larger dataset (NYT), the baseline setting (CPU) method runs 396s for one iteration on average while the Doc-distribution method (GPU) takes 12.0s. The Matrix-based method cannot handle the large memory required by NYT dataset. Detailed results are shown in the following tables.

Method & Machine	Time(s) of one iteration	Speedup
Lemur PLSA Implementation using CPU	1.560	1x
Doc-distribution method using GTX480 GPU	0.140	**11x**
Matrix-based method using GTX480 GPU	0.098	**16x**
State-of-the-art parallel method using 8-core CPU (expensive)	N/A	6x

Table 2: Speedup of PLSA with GPU using KOS dataset

Method & Machine	Time(s) of one iteration	Speedup
Lemur PLSA Implementation using CPU	396s	1x
Doc-distribution Method using GTX480 GPU	12s	**33x**
State-of-the-art parallel method using 8-core CPU (expensive)	N/A	6x

Table 3: Speedup of PLSA with GPU using NYT dataset

4 Comparison and Analysis

The state-of-the-art speedup method for PLSA is CPU-based [3] and can speed up by a factor of 6x with 8-core machine whose cost exceeds our GPU setting. Our GPU costs $450 GPU in market and our GPU-based methods deliver much more significant speedup. Regarding to the parallelization method, their method is close to our Matrix-based method and is memory intensive. This memory issue is addressed by our Doc-distribution method.

Our two methods emphasize different speedup factors. The Matrix-based method better exploits the GPU's parallism and fast in-block memory while the Doc-distribution method better accommodates the nature of PLSA and is more memory efficient. Their actual performance is affected by the nature of the datasets where the two most important factors are size and sparsity. On one hand, the size of the dataset would affect the applicability of the methods for two reasons. First, the device memory of GPU is limited to few gigabytes under the current infrastructure; second, the space complexity of the two methods are different: the Matrix-based method requires $O(DN \times TN)$ memory while the Doc-distribution method requires much a less $O(max(DN, TN) \times K)$ memory; on the other, the less sparse the doc-term count matrix is, the more speedup can the Matrix-based method achieve, since it better exploits the parallism and the fast shared memory but treats every doc-term pair as the same and ignores the fact that most of them are zero. Users can choose which speedup method to use with referring to these two factors of the dataset.

CONCLUSION AND FUTURE WORK

In this thesis, two methods are proposed to speed up PLSA via GPU, each contains a schema of workload parallelization, block-thread layout and memory optimization. The Doc-distribution method can process 300,000 documents in 12 seconds which is 33x speedup over the dominant Lemur-like implementation on a powerful CPU. The Matrix-based method achieves a slightly better speedup than the Doc-distribution method (16x and 11x) under smaller dataset. To conclude, our methods achieve the highest level of speedup given similar cost of machines among all related efforts, and this significant speedup can bring researchers in the text mining domain brand new experience. The major restriction of GPU-based methods is device memory, which limits the size of dataset our methods can process. Future work will involve utilizing CUDA's data streaming [15] mechanism to solve the issue of memory limit, as well as designing more efficient memory schema to utilize GPU's fast in-block memory and reduce the cost of simultaneous access to global variables, which happens to be part of PLSA and many other text mining algorithms.

APPENDIX

A hacky implementation of 'atomic' adding double number to global variable in kernel function:

```
union trans
{
    unsigned long long a;
    double b;
};

__global__ void
atomicAddDouble(double* address, const double adder)
{
        trans old_value, new_value;
        unsigned long long tmp;
        do {
                old_value.b = *address;
                new_value.b = old_value.b + adder;
                tmp = atomicCAS((unsigned long long*)address,
                        old_value.a, new_value.a);
        } while (tmp != old_value.a);
}
```

REFERENCES

[1] Y. Lu, Q. Mei, and C. Zhai, *Investigating task performance of probabilistic topic models: an empirical study of PLSA and LDA*, 2010.

[2] "Nvidia cuda zone," Tech. Rep., 2010.

[3] C. Hong, W. Chen, W. Zheng, J. Shan, Y. Chen, and Y. Zhang, "Parallelization and characterization of probabilistic latent semantic analysis," 2008.

[4] D. Newman, A. Asunction, P. Smith, and M. Welling, "Distributed inference for latent dirichlet allocation," 2007.

[5] W. Liao, "Parallel k-means data clustering," Tech. Rep. [Online]. Available: http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html

[6] M. Zechner and M. Granitzer, "Accelerating k-means on the graphic processor via cuda," 2009.

[7] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, "A parallel implementation of k-means clustering on gpus," 2008.

[8] H. Bai, L. He, D. Ouyang, Z. Li, and H. Li, "K-means on commodity gpus with cuda," 2009.

[9] L. Yu and Y. Xu, "A parallel gibbs sampling algorithm for motif finding on gpu," 2009.

[10] F. Yan, N. Xu, and Y. Qi, "Parallel inference for latent dirichlet allocation on graphic processing units," 2009.

[11] T. Hoffmann, "Probabilistic latent semantic analysis," 1999.

[12] A. Kaban, "Matlab implementation of plsa," Tech. Rep. [Online]. Available: http://www.cs.bham.ac.uk/~axk/ML_CODE/PLSA.m

[13] "Matrix multiplication, cuda samples, nvidia cuda zone," Tech. Rep., 2010.

[14] "Uci machine learning repository," Tech. Rep. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Bag+of+Words

[15] "Multi copy and compute, cuda samples, nvidia cuda zone," Tech. Rep., 2010.